

JPA (Hibernate) II.

Enum

- Mapování enum typů se provádí následujícím způsobem:

Enum typ:

```
public enum TypZakaznika {  
    FIRMA, OSVC, OBCAN  
}
```

Mapování v entitě:

```
@Enumerated  
private TypZakaznika typ;
```

- Standardně se do databáze uloží pořadové číslo vybraného typu:
 - FIRMA = 0, OSVC = 1, OBCAN = 2
- Je možné také do databáze ukládat přímo textový název enum typu:
 - `@Enumerated(EnumType.STRING)`
- Je možné i flexibilnější mapování, ale není zdaleka tolik komfortní:
 - <http://stackoverflow.com/questions/2751733/map-enum-in-jpa-with-fixed-values>
 - Od JPA 2.1 je možné toto: <https://dzone.com/articles/mapping-enums-done-right>

Embedded objekty I.

- Embedded objekt nemá vlastní identitu a uloží se do databáze jako součást entity.

Entita:

```
@Entity
public class Customer {
    @Embedded
    private Address address;
    //...
}
```

Embedded objekt:

```
@Embeddable
public class Address {
    private String city;
    private String street;
    @Column(name = "postal_code")
    private String zip;
    //...
}
```

Embedded objekty II.

- Protože embedded třídy tvoříme pro jejich znovupoužití, je občas nutná větší flexibilita mapování přepsáním výchozího mapování embedded třídy v entitě:

Další entita, která používá třídu Address:

```
@Entity
public class Company {
    @Embedded
    @AttributeOverrides(value = {
        @AttributeOverride(name = "zip",
            column = @Column(name = "zip"))
    })
    private Address address;
    //...
}
```

Embedded objekty III.

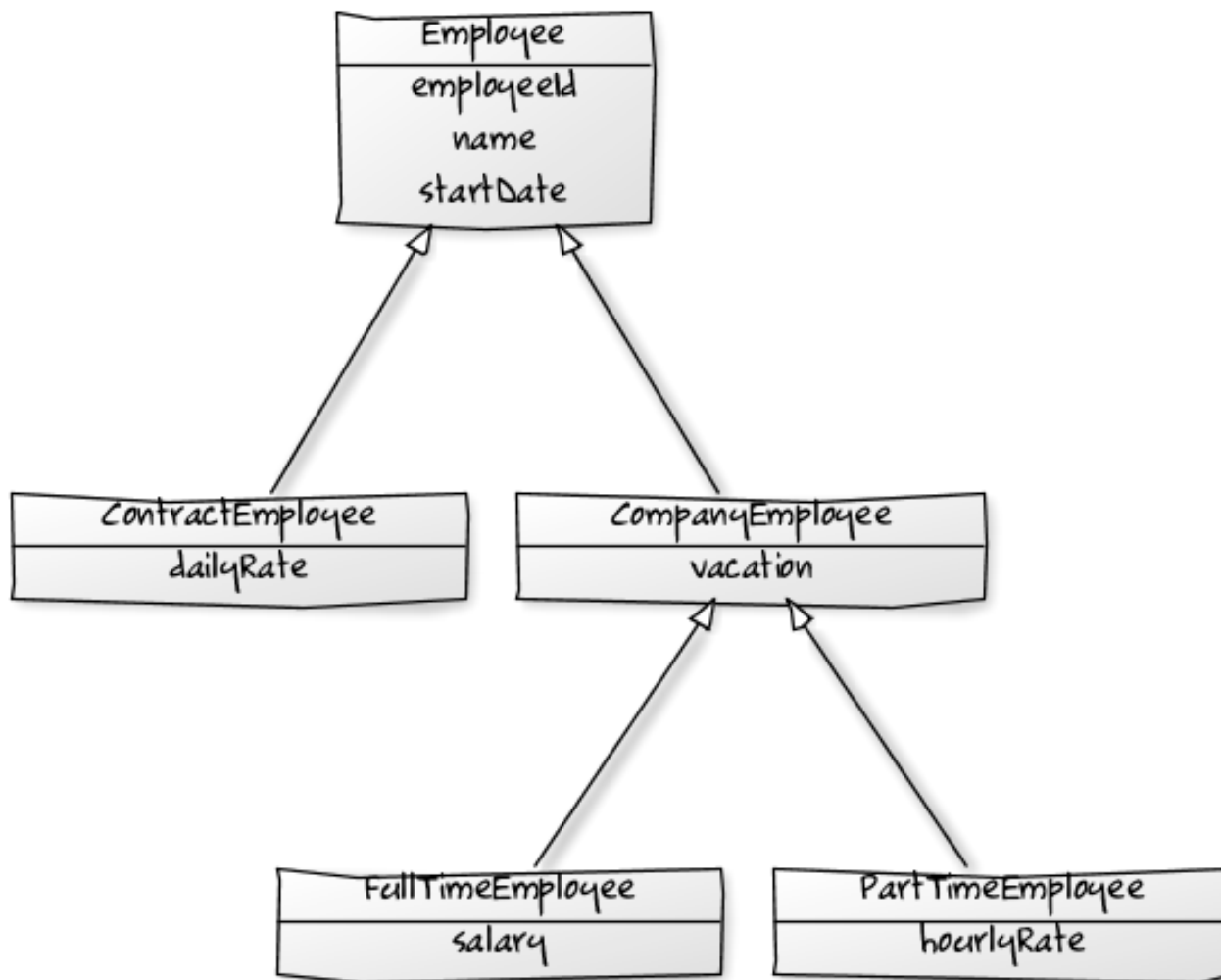
- Od JPA 2.0 mohou embedded třídy obsahovat vnořené embedded třídy a vazby na entity.
- V entitách, které obsahují embedded objekty, je možné nejenom předefinovat názvy sloupců, ale i celé vazby pomocí anotací `@AssociationOverrides` a `@AssociationOverride`.

Dědičnost

- Je několik typů dědičnosti v Hibernate, ale v základu se dá dozdělit dvou základních kategorií:
 - Vrcholem hierarchie je `@MappedSuperclass`
 - Dědičnost není v databázi zachycena jakýmkoli způsobem, nachází se pouze v doménovém modelu.
 - Příklad: Ve spoustě aplikací je předkem všech entit třída (s anotací `@MappedSuperclass`), která obsahuje atributy (které se namapují na sloupce): id a datum přidání záznamu do databáze. Tyto sloupce jsou poté v každé tabulce v databázi.
 - Vrcholem hierarchie NENÍ `@MappedSuperclass`

Mapování dědičnosti I.

- Budeme mapovat následující diagram tříd:



Mapování dědičnosti II.

- Všechny třídy, které mají být entity označíme anotací `@Entity`. Pokud si nepřejeme aby nějaká třída byla entitou (v tomto případě třída `CompanyEmployee`), neoznačíme ji anotací `@Entity`, ale anotací `@MappedSuperclass`.
 - Takové třídy se obvykle také označují klíčovým slovem `abstract`, aby vůbec nebylo možné vytvořit jejich instanci.
 - Takovou třídu navíc není možné označit anotací `@Table`.
 - Také není možné na ni odkazovat pomocí vazby z jiné entity.
- Třída v dědické hierarchii která by nebyla označena ani jednou z výše uvedených anotací se nazývá **transient třída**. Její stav je součástí entity, ale není spravován persistence providerem! Takové třídy se také obvykle označují klíčovým slovem `abstract`.

Mapování dědičnosti III.

@Entity

```
public class Employee {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "EMPLOYEE_ID")
```

```
    private Long employeeId;
```

```
    private String name;
```

```
    @Temporal(TemporalType.DATE)
```

```
    @Column(name = "START_DATE")
```

```
    private Date startDate;
```

```
    // TODO add getters and setters
```

```
}
```

Mapování dědičnosti IV.

@Entity

```
public class ContractEmployee extends Employee {  
    @Column(name = "DAILY_RATE")  
    private int dailyRate;  
    // TODO add getters and setters  
}
```

@MappedSuperclass

```
public abstract class CompanyEmployee extends Employee {  
    private int vacation;  
    // TODO add getters and setters  
}
```

Mapování dědičnosti V.

@Entity

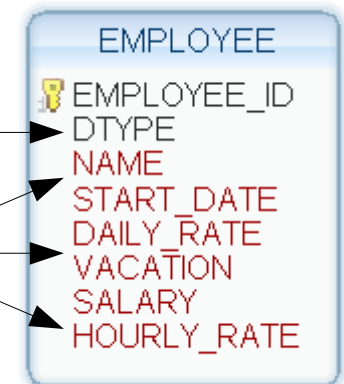
```
public class FullTimeEmployee extends CompanyEmployee {  
    private long salary;  
    // TODO add getters and setters  
}
```

@Entity

```
public class PartTimeEmployee extends CompanyEmployee {  
    @Column(name = "HOURLY_RATE")  
    private long hourlyRate;  
    // TODO add getters and setters  
}
```

Strategie mapování dědičnosti I.

- Existují tři strategie mapování dědičnosti. Strategie, která se použije se nastaví na entitě, která je kořenem hierarchie anotací `@Inheritance`:
 - Single table strategie:**
 - `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`
 - Výchozí strategie mapování. Není nutné anotaci `@Inheritance` vůbec uvádět.
 - Vytvoří se jedna tabulka, ve které jsou všechny sloupce ze všech entit v hierarchii a jeden sloupec s názvem `DTYPE`, ve kterém je název typu entity.
 - Sloupce, které nejsou v příslušném typu obsaženy, mají hodnotu `NULL`. Z toho vyplývá horší správa integritních omezení na úrovni databáze.
 - Tento typ strategie zabírá více místa v databázi, ale operace (čtení i zápis) jsou velice rychlé, protože nevyžadují spojování tabulek.



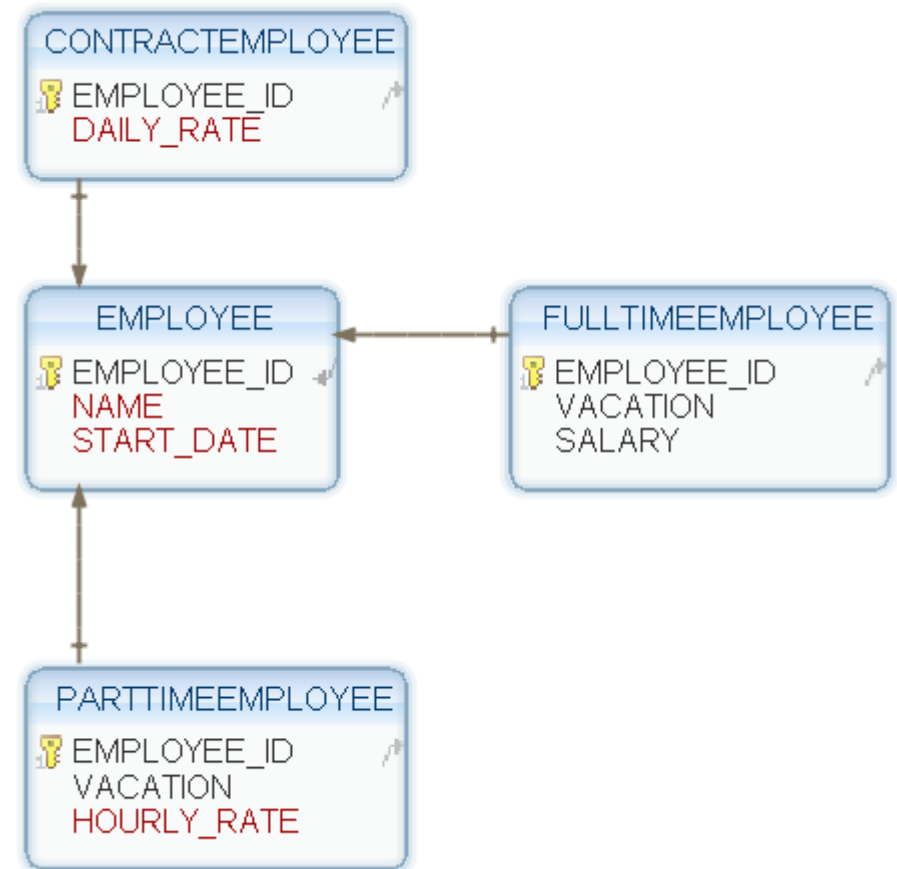
Strategie mapování dědičnosti II.

- Sloupec s názvem DTYPE se nazývá discriminator.
- Název tohoto sloupce je možné změnit:
 - `@DiscriminatorColumn(name="EMP_TYPE")`
- Hodnota tohoto sloupce je standardně textový název třídy. Sloupec může nabývat následujících hodnot: STRING, INTEGER, CHAR.
- Hodnotu sloupce je možné nastavit pro jednotlivé entity pomocí anotace:
 - `@DiscriminatorValue(value="EMP")`

Strategie mapování dědičnosti III.

- Další strategií mapování dědičnosti je **joined strategy**:

- `@Inheritance(strategy=InheritanceType.JOINED)`
- Pro každou entitu se vytvoří tabulka.
- Umožňuje normalizované tabulky v databázi, v databázi zabírají méně místa než při použití single table strategie.
- Operace (čtení i zápis) jsou pomalejší než u single table strategie.

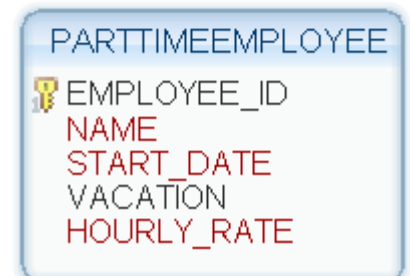
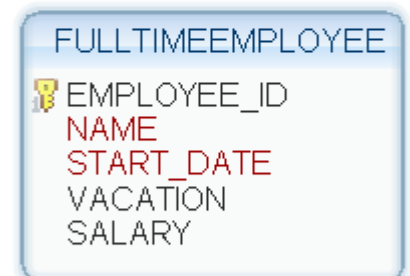
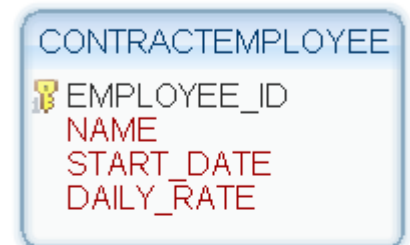


Strategie mapování dědičnosti IV.

- **Joined strategie – pokračování:**
 - Tabulky jsou pospojované prostřednictvím primárního klíče (PK) `EMPLOYEE_ID`, který je ve třídě `EMPLOYEE`. V dalších tabulách plní roli primárního-cizího klíče (PFK).
 - U entit je možné použít anotaci `@Table` pro změnu výchozího názvu tabulek.
 - Čím je hierarchie tříd širší nebo hlubší, tím je nutné více spojení (join) u `SELECT` operací.

Strategie mapování dědičnosti V.

- Poslední je strategie **Table-per-Concrete-Class**. U této strategie má každá entita svoji tabulku, ve které má veškerý svůj stav.
- Tuto strategii nemusí implementovat všechny JPA implementace, ale Hibernate ji podporuje.
- U této strategie je v současné době možné použít pouze následující způsoby generování primárního klíče: SEQUENCE, TABLE.
- Nevýhoda použití této strategie je v tom, že polymorfní dotazy jsou více náročnější než u jiných strategií.
- Výhoda použití této strategie je v tom, že je rychlejší získávání jednoho typu entity protože se fyzicky jedná o jednu tabulku a tudíž nedochází ke spojování tabulek při SELECTu. Další výhodou je, že se nepoužívá discriminator sloupec.



Strategie mapování dědičnosti VI.

- **Table-per-Concrete-Class pokračování:**
 - Je možné používat anotace `@AttributeOverride`, `@AssociationOverride` a `@Table`.
- Je také možné kombinovat různé typy dědičnosti v rámci jedné dědické hierarchie (**mixed inheritance**).

Mapování: field vs. property

- Je možné mít mapování na fieldech (atributech) nebo properties (getterech). Který typ použít? Jaké jsou výhody / nevýhody?
- Není možné tyto dva přístupy kombinovat, je nutné vybrat jeden z nich.
- Mapování na attributech:
 - Čitelnější, ale méně flexibilní.
- Mapování na getterech:
 - Flexibilnější, ale více nepřehledné (konfigurace je rozeseta po celé třídě).

Hibernate & Spring

- Při konfiguraci Hibernate a Spring jsou dvě možnosti:
 - Buď pomocí JPA (moderní, preferovaný způsob)
 - Nebo pomocí Hibernate (starší, Hibernate-specific způsob)

Spring & Hibernate: Spring konfigurace I.

```
<bean id="myDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

```
  <property name="url" value="jdbc:hsqldb:hsqldb://localhost/test" />
```

```
  <property name="username" value="sa" />
```

```
  <property name="password" value="" />
```

```
</bean>
```

Na produkci je datasource obvykle definovaný v Java EE serveru, přístup je k němu poté přes JNDI:

```
<jee:jndi-lookup id="myDataSource"
  jndi-name="java:comp/env/jdbc/myds"/>
```

```
<bean id="mySessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
```

```
  <property name="dataSource" ref="myDataSource" />
```

```
  <property name="configLocation" value="classpath:hibernate.cfg.xml" />
```

```
  <property name="hibernateProperties">
```

```
    <value>
```

```
      hibernate.dialect=org.hibernate.dialect.HSQLDialect
```

```
    </value>
```

```
  </property>
```

```
</bean>
```

Pozor! Používá se
Hibernate 4!
Pro Hibernate 3
stačí jenom změnit
toto číslo verze.

Je možné použít
externí konfiguraci

Je možné zde dodefinovat vlastní konfiguraci.
Zároveň je možné předefinovat konfiguraci,
která byla definována v hibernate.cfg.xml

Spring & Hibernate: Spring konfigurace II.

- Zapnutí podpory transakcí, které budou řízeny anotacemi:

```
<context:component-scan base-package="com.test" />
```

← V jakém balíčku (a podbalíčcích) jsou třídy s anotacemi

```
<bean id="transactionManager"  
      class="org.springframework.orm.hibernate4.HibernateTransactionManager">
```

↙ Pozor! Hibernate 4!

```
<property name="sessionFactory" ref="mySessionFactory" />
```

```
<property name="dataSource" ref="myDataSource" />
```

```
</bean>
```

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

↙ Zapnutí transakcí pomocí anotací

Spring & Hibernate: Servisní vrstva

Všechny metody této třídy budou transakční

- Servisní třída:

@Service

@Transactional

public class AppService {

@Autowired

private SessionFactory sessionFactory;

@Transactional(readonly = **true**)

@SuppressWarnings("unchecked")

public List<Employee> getEmployees() {

return sessionFactory.getCurrentSession().createQuery("from Employee").list();

}

public void addEmployee(Employee employee) {

 sessionFactory.getCurrentSession().persist(employee);

}

}

← @Transactional je možné uvést před názvem třídy, pak mají podporu transakcí všechny metody. Zároveň veškeré nastavení, které se zde uvede je výchozí pro všechny metody. Na úrovni metod je možné ho předefinovat.

← U metod, které nemění stav databáze je vhodné přidat @Transactional(readonly = **true**)

Šíření transakcí (propagation) I.

- Definuje způsob šíření transakcí, když jedna transakční metoda zavolá ve svém kódu jinou (vnořenou) transakční metodu. Pro jednotlivé zanořované metody jsou vytvářeny logické transakce.
- REQUIRED** (výchozí nastavení) – všechny metody probíhají v jediné fyzické transakci v databázi (start – commit/rollback v databázi). Když logická transakce pro vnitřní metodu nastaví příznak rollbacku, vyvolá se výjimka `UnexpectedRollbackException`, která zabrání vnější transakční metodě, aby dál pokračovala v provádění svého kódu (celá transakce bude zrušena).

REQUIRED

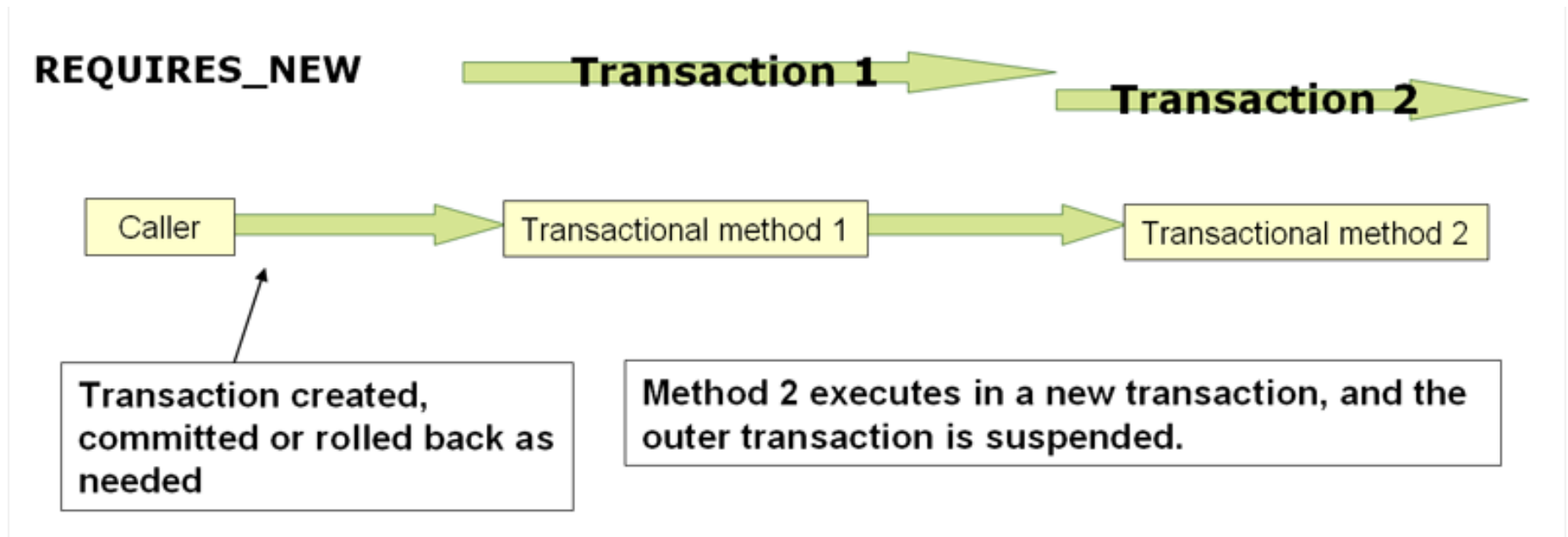


Transaction created,
committed or rolled back as
needed

Method 2 executes in the existing transaction.

Šíření transakcí (propagation) II.

- **REQUIRES_NEW** – pro každou transakční metodu je vytvořena samostatná fyzická transakce v databázi. Vnější logická (a zároveň fyzická) transakce může provést commit nebo rollback nezávisle na způsobu ukončení vnitřní transakce. Toto nastavení umožňuje vnější metodě pokračovat v transakci (se šancí na commit), i když logická (a fyzická) transakce pro vnitřní metodu skončila rollbackem (vnější metoda běží v jiné fyzické transakci).



Rollback

- Standardně se transakce zruší když se uvnitř transakční metody vyhodí výjimka typu `RuntimeException`. Když se vyhodí jiná výjimka (například `IOException`), tak se transakce potvrdí!
- Toto výchozí chování je možné změnit:
 - Rollback se provede pro výjimky typu `RuntimeException`, `IOException` a `SQLException`:
`@Transactional(rollbackFor={IOException.class, SQLException.class})`
 - Výjimky typu `RuntimeException` se budou vždy odchytávat (pokud je atributem `noRollbackFor` nevyloučíte).
 - Zajímavé atributy anotace `@Transactional` jsou:
 - `rollbackFor`, `noRollbackFor`, `timeout`, `readOnly`

Isolation levels

- Jsou k dispozici čtyři úrovně izolace:
 - **READ_UNCOMMITTED** – aktuální transakce může vidět data, která ještě nejsou potvrzena v databázi (jsou uncommitted).
 - **READ_COMMITED** – aktuální transakce nemůže vidět uncommitted data. **Výchozí strategie.**
 - **REPEATABLE_READ** – aktuální transakce nemůže vidět uncommitted data a zabrání se non-repeatable reads.
 - **SERIALIZABLE** – zabrání se non-repeatable reads, dirty-reads, phantom reads.
- Nastavení úrovně izolace:

```
@Transactional(isolation=Isolation.UROVEN_IZOLACE)
```


Spring & Hibernate / JPA: Transakce pomocí anotací I.

- Alternativně je možné k řízení transakcí nepoužívat anotace (v třídách nebude anotace `@Transactional` a v konfiguraci nebude tag `<tx:annotation-driven />`):

```
<aop:config>
```

```
  <aop:pointcut id="serviceMethods"  
    expression="execution(* com.test.service.*(..))" />
```

Balíček, ve kterém jsou třídy,
které přistupují do databáze



```
  <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethods" />
```

```
</aop:config>
```

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
```

```
  <tx:attributes>
```

```
    <tx:method name="*" propagation="REQUIRED" />
```

```
  </tx:attributes>
```

```
</tx:advice>
```

Spring & Hibernate / JPA: Transakce pomocí anotací II.

- `HibernateTransactionManager` a `JtaTransactionManager` spravují transakce pouze v rámci jedné aplikace. V případě, že máte na Java EE serveru JTA, pak je možné použít `JtaTransactionManager`.
- Změna je pouze v konfiguraci, není nutné měnit kód.

Metody třídy Session I.

- Metody pro změnu stavu entity: **transient** → **persistent**
 - **persist()**
 - Není garance okamžitého přidělení identifikátoru (ID). Přidělení ID se může provést až při provedení flush.
 - Garantuje, že se neprovede INSERT příkaz když je tato metoda zavolána mimo transakci.
 - **save()**
 - Garantuje okamžité přidělení identifikátoru (ID).
 - Když se má zavolat DB operace INSERT pro přidělení identifikátoru (pomocí generátoru IDENTITY), poté se INSERT zavolá ihned a nezjišťuje se, jestli jsme aktuálně uvnitř nebo mimo transakci.
- Když ukládáte entitu, která má kolekci entit spolu s prvky z této kolekce, pak je možné je persistovat v jakémkoli pořadí pokud nemají NOT NULL omezení na cizím klíči. Toto omezení ale můžete porušit, když voláte operaci `save()` v nesprávném pořadí.

Metody třídy Session II.

- Metody pro získání entity podle jejího primárního klíče:

- **load()**

```
Employee employee = (Employee)session.load(Employee.class, new Long(1));
```

- Když taková entita neexistuje, pak se vyhodí výjimka.
 - Pokud je instance entity ve 1st level cache, pak se nezavolá SELECT do databáze, ale vrátí se entita z cache.

- **get()**

- Pokud si nejste jisti, jestli entita s hledaným klíčem existuje. Pokud neexistuje, vrátí null.
 - SELECT do databáze se zavolá pokaždé.

Metody třídy Session III.

- Je možné kdykoli vynutit načtení entity z databáze:
 - **refresh()**
 - Vhodné, když databázové triggery inicializují nějaká data v databázi:

```
// ulozeni entity do databaze
```

```
session.save(employee);
```

```
// vynuceni SQL INSERTu
```

```
session.flush();
```

```
// nacte aktualni data z databaze (po vykonani triggeru)
```

```
session.refresh(employee);
```

Metody třídy Session III.

- Pro low-level práci s SQL je užitečná metoda doWork() (i když JdbcTemplate ze Springu je stále lepší):

```
session.doWork( connection -> {  
    try(Statement statement = connection.createStatement()) {  
        statement.executeUpdate( "UPDATE person SET name = UPPER(name)" );  
    }  
} );
```


Metody třídy Session IV.

- K získání dat z databáze se obvykle používají dotazy (query). Existují čtyři typy dotazů, které Hibernate podporuje:
 - HQL (Hibernate Query Language)
 - Criteria Query
 - Example Query
 - Native SQL Query

Metody třídy Session V.

- **createQuery():** Metoda pro vykonávání HQL dotazů
 - Na této metodě můžete vykonávat následující metody:

```
Employee employee = (Employee) session
```

```
.createQuery("from Employee where manager is null")
```

```
.uniqueResult();
```

← Vráť jeden objekt. Pokud nebyl žádný objekt nalezen, vrátí null. Pokud dotaz vrací více objektů, vyhodí se chyba.

```
List<Employee> employees = session.createQuery("from Employee").list();
```

↑ Vráť list objektů

Metody třídy Session VI.

- Další metody `createQuery()`:
 - Je možné přidávat parametry do query buď přes index jako v JDBC nebo přes pojmenované parametry:

```
List<Employee> employees =  
    session.createQuery("from Employee where address.city = :city")  
        .setString("city", city).list();
```

- Parametry mohou být různých typů: například `String`, `Integer`, `Date`, ale i entita.
- Jako návratový typ nemusí být jenom jedna entita, ale i více entit nebo jakýkoli objekt:

```
long count = (long) session  
    .createQuery("select count(e) from Employee e").uniqueResult();
```

Metody třídy Session VII.

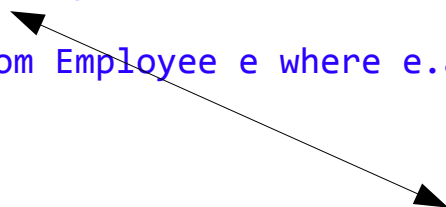
- Další metody `createQuery()`:
 - Podpora pro stránkování:
 - `setFirstResult(int)`: nastaví číslo řádku, od kterého se budou záznamy vracet.
 - `setMaxResults(int)`: nastaví maximální počet výsledků, které dotaz vrátí.
 - Nezapomeňte Váš dotaz před jeho „ořezáním“ pomocí těchto dvou metod utřídit pomocí `order by`

Metody třídy Session VIII.

- Je vhodné nemít HQL dotazy ad-hoc v kódu, ale uložit je buď do externích XML souborů (obvykle s příponou `hbm.xml`) nebo k entitám pomocí anotací `@NamedQuery` a `@NamedQueries`:

- Definice named query (u entity):

```
@NamedQuery(name="Employee.withCity",  
            query="select e from Employee e where e.address.city = :city")
```



- Použití named query:

```
List<Employee> employees = session.getNamedQuery("Employee.withCity")  
                                .setString("city", city).list();
```

- Definice named query je globální vůči session factory / entity manager factory. Proto je vhodné zavést vlastní konvenci pojmenování například jako na uvedeném příkladě.
- Named query je oproti ad-hoc query rychlejší na vykonávání, protože se parsuje při startu a ne až za běhu.

Metody třídy Session IX.

- Pomocí metody `createSQLQuery()` je možné vykonat nativní SQL dotaz.
- Nativní SQL dotaz je možné obdobně jako HQL dotaz externalizovat pomocí anotací `@NamedNativeQuery` a `@NamedNativeQueries`. Je také možné uložit ho do externího souboru (obvykle s příponou `hbm.xml`).

Metody třídy Session X.

- Uvnitř transakce je možné libovolně modifikovat persistentní entity, přičemž veškeré změny se uloží do databáze když se zavolá flush operace (není nutné ji volat ručně, volá se automaticky při ukončení transakce).
- Velice často se ale dostanete do situace, kdy načtete objekt, session se ukončí, následně potřebujete změnit stav entity a uložit ji do databáze (pomocí jiné session). Jak na to?
 - `update()`: Když jste si jisti, že session neobsahuje objekt se stejným identifikátorem (čili v úplně nové, nově vytvořené session).
 - `merge()`: Když chcete spojit změny v entitě nezávisle na stavu session.
- V Hibernate také existuje metoda `saveOrUpdate()`, která vyvolá funkci `save()`, pokud entita nemá identifikátor a funkci `update()`, pokud entita identifikátor má.

Metody třídy Session XI.

- Metoda `delete()` smaže entitu z databáze. Obecně můžete mazat entity v jakémkoli pořadí (v rámci jedné transakce), ale můžete také porušit NOT NULL omezení na cizím klíči.
- **Flush** je proces, který synchronizuje stav databáze se stavem objektů v paměti.
 - Kdy se vykonává?
 - Před vykonáním některých dotazů
 - Při potvrzení transakce (commit)
 - Při explicitním zavolání metody `flush()`
- Kromě explicitního zavolání této metody není možné říct kdy se vyvolá, jenom je pevně dané pořadí operací, které při něm probíhají. Každopádně Hibernate garantuje, že metoda `list()` nikdy nevrátí chybná data.
- Můžete změnit chování flush procesu:
`session.setFlushMode(FlushMode.MANUAL);`

Flush módy:

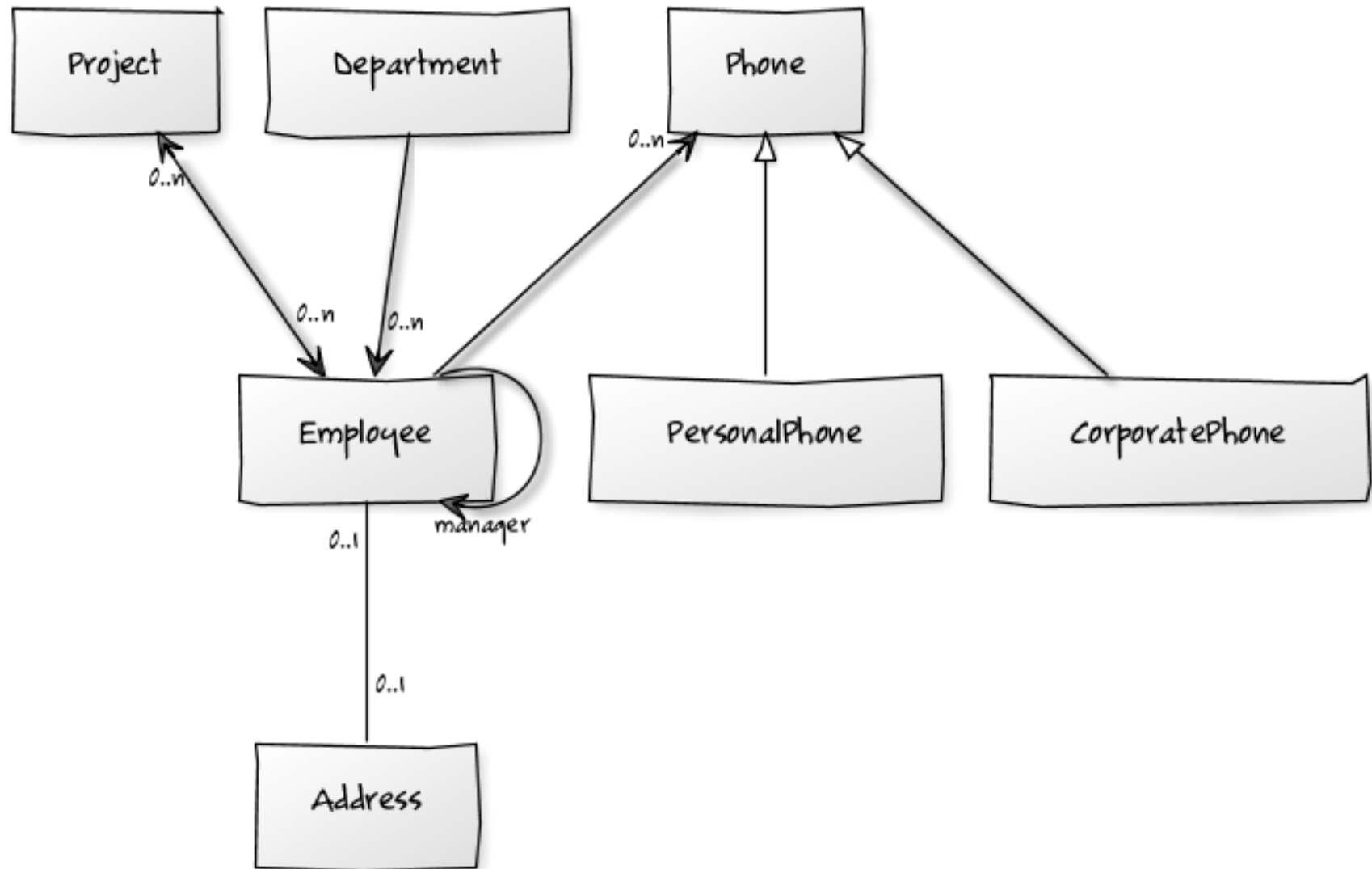
- **AUTO**: defaultní mód
- **ALWAYS**: po každém dotazu, velice neefektivní
- **COMMIT**: pouze po potvrzení transakce
- **MANUAL**: pouze když se explicitně zavolá `flush()`.
Může být velice efektivní.

Metody třídy Session XII.

- Při flush procesu se vykonávají následující operace v tomto pořadí:
 - 1) INSERT entit ve stejném pořadí, v jakém byly uloženy pomocí metody `save()`.
 - 2) UPDATE entit.
 - 3) DELETE kolekcí.
 - 4) DELETE, UPDATE a INSERT elementů kolekcí.
 - 5) INSERT kolekcí.
 - 6) DELETE entit ve stejném pořadí, v jakém byly smazány použitím metody `delete()`.

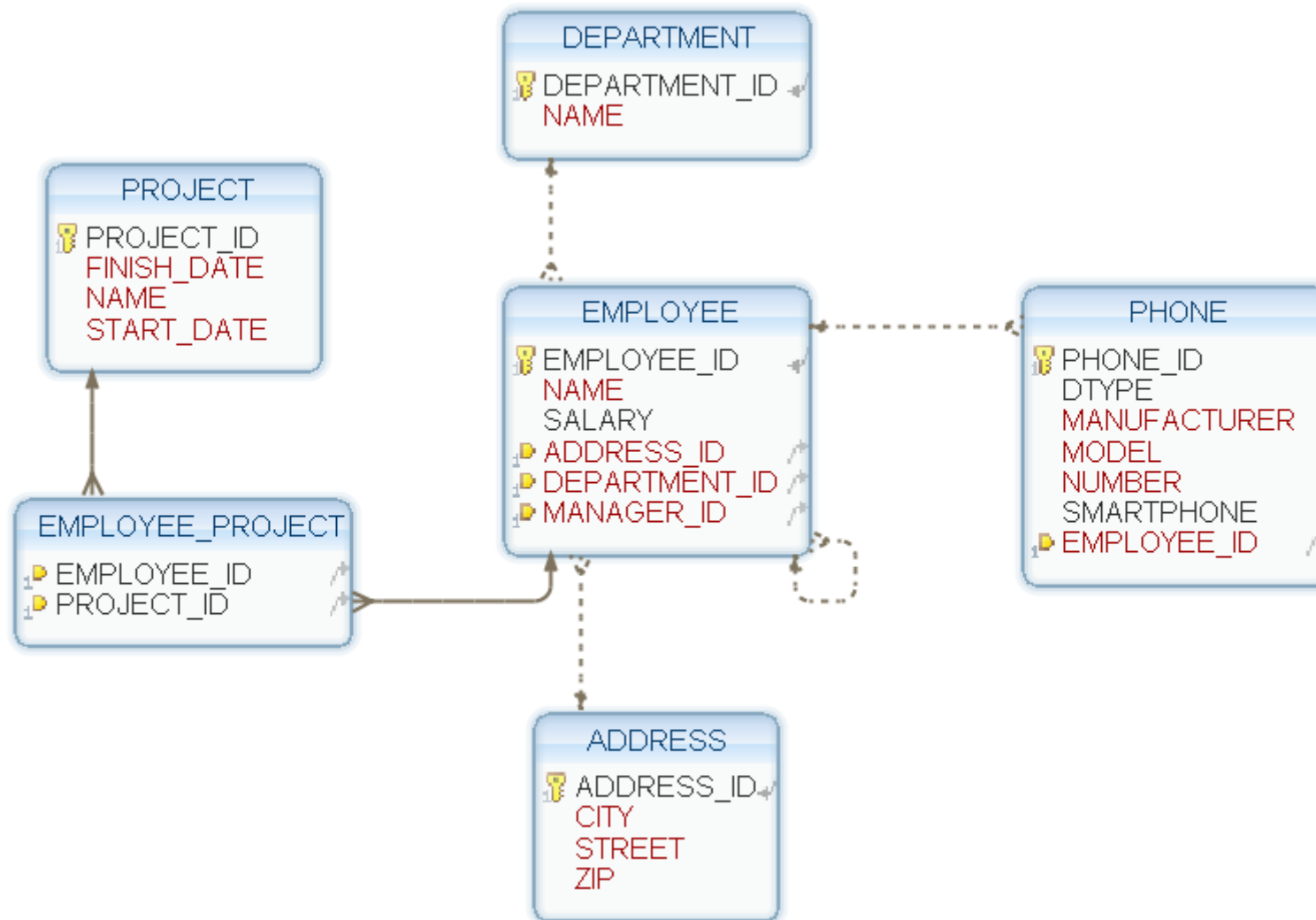
Databáze – diagram tříd

- Máme databázi s následujícím diagramem tříd:



Databáze – ER diagram

- Pro pořádek uvedu ještě ER diagram:



HQL: case sensitivity, klauzule FROM

- Kromě názvů Java tříd a jejich atributů jsou dotazy case-insensitive.
 - `select = Select = SELECT`
 - `com.test.Customer != com.test.customer`
- Nejjednodušší HQL query:
 - `from com.test.entity.Employee` – Vrátí všechny instance třídy `com.test.entity.Employee`
 - Ještě jednodušší varianta: `from Employee`
- Když se potřebujeme odkázat na entitu z jiné části dotazu, pak je nutné přiřadit alias:
 - `from Employee as emp`
 - Klíčové slovo `as` je nepovinné, můžete také napsat: `from Employee emp`

HQL: vazby a JOIN

- **Kartézský součin (cross join):**
 - `from Employee, Project`
- **Inner join** (klíčové slovo inner je nepovinné):
 - Získá všechny zaměstnance a všechny jejich telefony:
 - `from Employee emp inner join emp.phone ph`
- **Left outer join, Right outer join** (klíčové slovo outer je nepovinné):
 - Získá všechny oddělení a zaměstnance, kteří jsou v nich zaměstnaní (v nějakém oddělení nemusí být žádní zaměstnanci):
 - `from Employee emp right outer join emp.department dep`
- **Full join**

HQL: vazby a JOIN

- Je možné přidat více podmínek pro spojení použitím klíčového slova **with**:
 - Získá všechny oddělení ve firmě, přičemž vypustí z vazby zaměstnance, kteří pracují níž než ve druhém patře (jejich hodnota bude null):

```
from Employee emp right outer join emp.department dep with dep.buildingFloor >= 2
```
- **Fetch join** umožňuje, aby byly asociace nebo kolekce hodnot inicializovány společně s jejich otcovskými objekty použitím jednoho SELECTu. Efektivně se tím přepíše lazy deklarace mapování pro vazby a kolekce:
 - ```
from Employee emp left join fetch emp.subordinates sub
```
- Spojením více než jedné kolekce pomocí join fetch se může vytvořit kartézský součin.

# Optimalizace SQL dotazu I.

- Tento HQL dotaz vygeneruje minimálně n+1 SQL dotazů:

```
from Employee
```

- Proč? Protože máme vazby OneToOne a ManyToOne ve výchozím nastavení: `fetch=FetchType.EAGER`
- Abyste tomuto problému zabránili, je vhodné u všech vazeb typu OneToOne a ManyToOne nastavit: `fetch=FetchType.LAZY`
- Poté je nutné pro dotažení dat z databáze použít v HQL `join fetch`:

```
from Employee emp left join fetch emp.department
```

- Nebo u criteria:

```
session.createCriteria(Employee.class)
 .setFetchMode("department", FetchMode.JOIN).list();
```

# Optimalizace SQL dotazu II.

- OneToOne vazba je při optimalizaci problematictější než ManyToOne:
- Nastavte na vazbě OneToOne u entit Employee i Address  
fetch=FetchType.LAZY
- Tento dotaz vygeneruje n+1 SQL dotazů:

```
from Employee e left join fetch e.address
```

- Tento dotaz vygeneruje 1 SQL dotaz:

```
from Address a left join fetch a.employee
```

- Proč se Hibernate chová jinak? Kvůli nastavení JoinColumn a mappedBy.
- Ultimátní řešení? Místo dvou vazeb OneToOne použít OneToMany ↔ ManyToOne.
  - <http://www.bigsoft.co.uk/blog/index.php/2008/10/26/lazy-loading-one-to-one-relationships>



# Optimalizace SQL dotazu III.

- Join fetch je možné řetězit:

```
from Employee e left join fetch e.department left join fetch e.manager
```

- Není možné v tomto SELECTu kombinovat všechny typy vazeb, ale OneToOne a ManyToOne vazby fungují vždy.
- Vždy se podívejte na výsledné SQL.

# HQL: vazby a JOIN

- Pokud používáte lazy fetching na úrovni property, pak je možné přinutit Hibernate aby načetl lazy properties pomocí `fetch all properties`:
  - `from Document fetch all properties`
- Často je možné použít implicitní join (bez klíčového slova join):
  - Získá všechny zaměstnance, kteří bydlí v Praze:
  - `from Employee emp where emp.address.city = 'Prague'`
- Máte dvě možnosti jak získat hodnotu primárního klíče entity:
  - Pomocí speciální property `id` (musí být malými písmeny) pokud entita nemá atribut `id`, který není primárním klíčem.
  - Vždy je možné použít název atributu, který je primárním klíčem.

# HQL: klauzule SELECT

- SELECT klauzule vybírá které objekty a jejich atributy budou ve výsledném result setu.
  - Vrátí jména zaměstnanců:
    - `select emp.name from Employee emp`
- Dotazy mohou vracet atributy tranzitivně.
  - Vrátí město bydliště zaměstnanců:
    - `select emp.address.city from Employee emp`
- Dotazy mohou vracet více objektů anebo properties:
  - Jako pole typů `Object[]`:
    - `select emp.name, emp.address.city from Employee emp`
  - Jako pole typů `List`:
    - `select new list(emp.name, emp.address.city) from Employee emp`
  - Jako pole uživatelských typů (za předpokladu, že třída `EmployeeDetail` má příslušný konstruktor):
    - `select new EmployeeDetail(emp.name, emp.address.city) from Employee emp`

# HQL: klauzule SELECT, agregační funkce

- Vybraným výrazům je možné přiřadit alias použitím klíčového slova **as**:
  - **select** max(salary) as max, min(salary) as min **from** Employee emp
  - Velice užitečné to je ve spojení se **select new map**:
  - **select new** map (max(salary) as max, min(salary) as min) **from** Employee emp
- Agregační funkce:
  - avg(...), sum(...), min(...), max(...), count(\*), count(distinct ...), count(all ...), count(...)
    - ↑ Spočítá i NULL řádky
    - ↑ Vyřadí duplicitní řádky
    - ↙ To samé, vyřadí NULL řádky
- V **SELECT** klauzuli je možné používat aritmetické operátory, spojení textů a SQL funkce:
  - **select upper**(emp.name) || ' has salary ' || emp.salary || ', - Kc' **from** Employee emp

# HQL: Polymorfní dotazy

- Následující dotaz: `from Phone as p` vrací instance nejenom třídy `Phone`, ale také všech jejích potomků.
- V HQL můžete uvést ve `from` klauzuli libovolnou třídu nebo interface. Dotaz vrátí instance všech persistentních tříd, které jsou potomky takové třídy nebo implementují příslušný interface.
- Následující dotaz získá všechny persistentní objekty:
  - `from java.lang.Object`
- Vzhledem k tomu, že obdobné dotazy fyzicky vykonávají více `SELECTů` do databáze, `ORDER BY` klauzule neutřídí správně celý result set!

# HQL: klauzule WHERE

- Where klauzule umožňuje omezit počet instancí vracených SELECTem:
  - Bez aliasu: `from Employee where name = 'Adam Boss'`
  - S aliasem: `from Employee emp where emp.name = 'Adam Boss'`
- Je možné navigovat v objektovém grafu:
  - `from Employee emp where emp.address.city = 'Prague'`
- Operátor = je možné použít nejen k porovnání properties, ale i instancí.
- Speciální property class slouží k porovnání instancí:
  - `from Phone p where p.class = CorporatePhone`

# HQL: výrazy ve WHERE podmínce I.

- Matematické operátory: +, -, \*, /
- Operátory porovnání: =, >=, <=, <>, !=, like
  - Vybere zaměstnance, kteří mají ve jméně slovo „Boss“:  
`from Employee e where e.name like '%Boss%'`
- Logické operátory: and, or, not
- Závorky ( )
- in, not in, between, is null, is not null, is empty, is not empty, member of, not member of
  - Vybere oddělení ve firmě, která jsou mezi patry 2 až 5:  
`from Department d where d.buildingFloor between 2 and 5`
  - Vybere zaměstnance bez adresy: `from Employee where address is null`
  - Vybere všechny telefony, jejichž výrobcem je HTC nebo Nokia:  
`from Phone p where p.manufacturer in ('HTC', 'Nokia')`

# HQL: výrazy ve WHERE podmínce II.

- Vybere zaměstnance, kteří nemají žádný telefon:
  - `from Employee e where e.phones is empty`
- „Simple“ case: `case ... when ... then ... else ... end`
- „Searched“ case: `case when ... then ... else ... end`
- Spojení Stringů: `... || ...`
- `current_date()`, `current_time()`, `current_timestamp()`
  - Vybere plánované projekty v budoucnosti:  
`from Project p where p.startDate > current_date`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`,  
`month(...)`, `year(...)`
  - Vybere projekty, které začaly nebo skončily v roce 2011:  
`from Project p where year(p.startDate) = 2011 or year(p.finishDate) = 2011`



# HQL: výrazy ve WHERE podmínce III.

- Funkce a operátory definované EJB-QL 3.0:
  - `substring()`, `trim()`, `lower()`, `upper()`, `length()`,  
`locate()`, `abs()`, `sqrt()`, `bit_length()`, `mod()`
- `coalesce()`, `nullif()`
- `str()` pro konverzi numerických hodnot nebo datumů na String.
- `cast ( ... as ... )`, `extract ( ... from ... )` → pokud je ANSI `cast()` a `extract()` podporováno databází
- `index()`

# HQL: výrazy ve WHERE podmínce IV.

- Funkce, do kterých vstupuje jako argument kolekce: `size()`, `minelement()`, `maxelement()`, `minindex()`, `maxindex()`

Minimum a maximum  
z kolekce základních typů

Minimální a maximální  
index indexované kolekce

- Můžeme zjišťovat počet prvků kolekce následujícími způsoby:
  - **from** Employee e **where** e.projects.size = 0
  - **select** e.name **from** Employee e **where** size(e.projects) = 0
- Funkce jako `sign()`, `trunc()`, `rtrim()`, `sin()`
- `elements()`, `indices()` ve spojení s klíčovými slovy `some`, `all`, `exists`, `any`, `in`.
- Elementy indexovaných kolekcí můžete zpřístupnit pomocí [ ]

# HQL: klauzule ORDER BY, GROUP BY, HAVING

- Výsledky dotazu mohou být utříděny pomocí jakékoli property:
  - **from** Project p **order** by p.startDate **desc**, p.name
- Výsledek dotazu může být sgrupován podle jakékoli property:
  - Vypíše počty smartphonů a obyč. telefonů:
  - **select** p.smartphone as smartphone, count(p) as count **from** Phone p **group** by p.smartphone **order** by p.smartphone **desc**
- Je také možné použít having klauzuli:
  - Vypíše města, ve kterých žijí alespoň dva zaměstnanci:
  - **select** a.city as city, count(a) as employee\_count **from** Address a **group** by a.city **having** count(a) > 1

# HQL: Subquery

- Vybere zaměstnance, kteří mají nadprůměrný plat:
  - `from Employee emp where emp.salary > (select avg(salary) from Employee)`

# Query Hints

- Pomocí metody `setHint()` je možné nastavovat hinty. Jedním z nejzajímavějších je:
  - `QueryHints.HINT_PASS_DISTINCT_THROUGH`
    - Když se v JPA query volá `DISTINCT`, tak vyloučí duplicity v objektovém grafu, ale neposílá klíčové slovo `DISTINCT` do vygenerovaného SQL dotazu.
  - <http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/jpa/QueryHints.html>

# Stream

- Od Hibernate 5.2 Session nemusí vracet přímo list(), ale také stream()!

# Criteria Query I.

- Interface `Criteria` reprezentuje dotaz nad konkrétní persistentní třídou. Vytváří se z instance `Session`:
  - **HQL:** `from Employee`
  - **Criteria:** `session.createCriteria(Employee.class).list();`
- Získání entit:
  - `list()` vrací `java.util.List`
  - `uniqueResult()` vrací jeden `java.lang.Object`
- Třída `Restrictions` slouží k přidávání omezení:
  - **HQL:** `from Employee where name like '%Boss%'`
  - **Criteria:** `session.createCriteria(Employee.class).add(Restrictions.like("name", "%Boss%")).list();`
- Omezení je celá řada a odpovídají operátorům v HQL.

# Criteria Query II.

- Omezení je možné spojovat a vnořovat:
  - **HQL:** `select e from Employee e  
where salary > 30000  
and name not like '%Boss%'`
  - **Criteria:** `session.createCriteria(Employee.class)  
.add(Restrictions.gt("salary", new Long(30000)))  
.add(Restrictions.not(Restrictions.like("name", "%Boss%")));`
- Použití klíčového slova OR:
  - **HQL:** `select e from Employee e  
where e.name like '%A%'  
or e.name like '%B%'`
  - **Criteria:** `session.createCriteria(Employee.class).  
add(Restrictions.or(Restrictions.like("name", "%A%"),  
Restrictions.like("name", "%B%")));`
    - ↙ To samé ↘  
`Restrictions.like("name", "B", MatchMode.ANYWHERE)`



# Criteria Query III.

- Projections se používají k získání vybraných atributů:
  - **HQL:** `select salary from Employee`
  - **Criteria:** `session.createCriteria(Employee.class)  
 .setProjection(Property.forName("salary"));`
- Nebo pro práci s agregačními funkcemi:
  - **HQL:** `select avg(salary) from Employee`
  - **Criteria:** `session.createCriteria(Employee.class)  
 .setProjection(Projections.avg("salary"));`
- Náhrada klauzule GROUP BY:
  - **HQL:** `select manufacturer from Phone group by manufacturer`
  - **Criteria:** `session.createCriteria(Phone.class)  
 .setProjection(Projections.groupProperty("manufacturer"));`

# Criteria Query IV.

- Použití klíčového slova OR II. - můžeme také použít metodu disjunction:
  - **HQL:** `select model from Phone where manufacturer like 'HTC' or manufacturer like 'Nokia' group by model`
  - **Criteria:** `session.createCriteria(Phone.class)  
 .add(Restrictions.disjunction()  
 .add(Restrictions.like("manufacturer", "HTC"))  
 .add(Restrictions.like("manufacturer", "Nokia")))  
 .setProjection(Projections.projectionList()  
 .add(Projections.groupProperty("model"))  
 .add(Property.forName("model")));`

# Criteria Query V.

- Třídění:
  - **HQL:** `select name from Employee order by name`
  - **Criteria:** `session.createCriteria(Employee.class)  
 .addOrder(Order.asc("name"))  
 .setProjection(Property.forName("name"));`
- Použití kritérií v asociacích:
  - **HQL:** `select e from Employee e where e.address.city = 'Prague'`
  - **Criteria:** `session.createCriteria(Employee.class)  
 .createCriteria("address")  
 .add(Restrictions.eq("city", "Prague")).list();`
  - **Stejné kritérium pomocí aliasu:**
    - `session.createCriteria(Employee.class)  
 .createAlias("address", "a")  
 .add(Restrictions.eq("a.city", "Prague")).list();`

# Criteria Query VI.

- Je možné nastavit fetch asociace během runtime:
  - `session.createCriteria(Employee.class)`  
    `.setFetchMode("phones", FetchMode.JOIN).list();`
- Možné hodnoty:
  - **DEFAULT** - defaultní nastavení
  - **JOIN** - ekvivalent `fetch="join"`
  - **SELECT** - ekvivalent `fetch="select"`

# Detached Criteria

- Třída `DetachedCriteria` umožňuje vytvořit dotaz bez session a později ho vykonat, když je session k dispozici:
  - ```
DetachedCriteria detachedCriteria =  
    DetachedCriteria.forClass(Employee.class)  
    .add(Restrictions.isNull("manager"));  
    // ...  
detachedCriteria.getExecutableCriteria(session).list();
```
- Třída `DetachedCriteria` zároveň umožňuje vykonávání subquery:
 - **HQL:**

```
select e.salary from Employee e  
where e.salary > (select avg(salary) from Employee)
```
 - **Criteria:**

```
DetachedCriteria avgSalary =  
    DetachedCriteria.forClass(Employee.class)  
    .setProjection(Projections.avg("salary"));  
  
session.createCriteria(Employee.class)  
    .add(Property.forName("salary").gt(avgSalary))  
    .setProjection(Property.forName("salary"));
```

Example

- Pomocí třídy Example je možné vytvářet dotaz z instance třídy:

```
Employee emp = new Employee();
```

```
emp.setName("%Boss%");
```

```
Example example = Example.create(emp).excludeZeroes().enableLike();
```

```
session.createCriteria(Employee.class).add(example).list();
```

Filtry I.

- Od Hibernate 3 je možné vytvořit filtr kritéria a dynamicky je zapínat / vypínat na úrovni třídy nebo kolekce.
- Filtry plní stejnou funkci jako podmínky ve WHERE podmínce, ale je možné je dynamicky zapínat / vypínat.
- **Definice filtru na entitě:**

```
@Entity
@FilterDefs({
    @FilterDef(name = "employeeMinSalary", parameters = @ParamDef(name = "minSalary", type = "integer")),
    @FilterDef(name = "employeeMaxSalary", parameters = @ParamDef(name = "maxSalary", type =
"integer")) })
@Filters({
    @Filter(name = "employeeMinSalary", condition = "salary > :minSalary"),
    @Filter(name = "employeeMaxSalary", condition = "salary < :maxSalary") })
public class Employee {
    // ...
}
```

Filtry II.

- **Použití filtru:**

```
session.enableFilter("employeeMinSalary").setParameter("minSalary", 50000);  
session.enableFilter("employeeMaxSalary").setParameter("maxSalary", 90000);  
List<Employee> employees = session.createQuery("from Employee").list();
```

- Filtry je možné také vypínat:

```
disableFilter("nazevFiltru")
```

- Filtry mohou být také specifikované na vazbě:

Po zapnutí se
vyfiltrují projekty,
které mají finish_date
rovno NULL.

```
@Filter(name = "nonFinishedProjects", condition = "finish_date is not null")
```

```
private List<Project> projects;
```

Poznámka: Přidejte do
@FilterDefs tuto definici filtru:
@FilterDef(name = "nonFinishedProjects")

- Pomocí anotace @FilterJoinTable
je také možné filtrovat data přímo z
asociační tabulky

Fetch I.

- Fetch strategie spojení může být definována v metadatech O/R mapování, nebo předefinována příslušným dotazem.
- Defaultně používá Hibernate lazy select pro kolekce (mapování one-to-many, many-to-many) a eager fetching pro asociace s jednou hodnotou (mapování one-to-one, many-to-one).
- Přístup k lazy spojení mimo transakci způsobí výjimku! Hibernate neumožňuje lazy inicializaci pro detachované objekty!
- Často ale mimo transakci potřebujeme ke spojení přistoupit!
- Jednou z možností je toto nastavení, které ale pokaždé vyvolá spoustu SELECT dotazů na získání všech telefonů zaměstnance, i když chceme jenom pár informací ze zaměstnance:

```
public class Employee {  
    @OneToMany(mappedBy = "employee", fetch=FetchType.EAGER)  
    private List<Phone> phones;  
}
```

Fetch II.

- Tento HQL dotaz vrátí z databáze jeden řádek s objektem typu Employee:

```
select e from Employee e where e.manager is null
```

- **Použití:** Employee employee = (Employee) session.createQuery("from Employee where manager is null").uniqueResult();
- Když chceme mimo transakci získat informace o jeho telefonech, které jsou namapované pomocí anotace @OneToMany, vyhodí se výjimka typu LazyInitializationException.
- Tento HQL dotaz vrátí z databáze dva řádky, obsahující zaměstnance a jeho dva telefony. Pro spojení se použije LEFT OUTER JOIN:

```
select e, p from Employee e left join fetch e.phones p where e.manager is null
```

- **Použití:** Employee employee = (Employee) session.createQuery("select e, p from Employee e left join fetch e.phones p where e.manager is null").uniqueResult();

Fetch III.

- Získá jeden řádek s objektem typu Employee:

```
Employee employee = (Employee) session.createCriteria(Employee.class)
    .add(Restrictions.isNull("manager")).uniqueResult();
```

- Získá z databáze dva řádky s objektem typu Employee a jeho telefony (pomocí LEFT OUTER JOIN spojení):

```
Employee employee = (Employee) session.createCriteria(Employee.class)
    .add(Restrictions.isNull("manager"))
    .setFetchMode("phones", FetchMode.JOIN)
    .uniqueResult();
```

Fetch IV.

- Někdy nechcete inicializovat velkou kolekci, ale stále potřebujete nějaké informace o ní, jako její velikost, nebo část jejich dat. K tomu můžete použít collection filter.
- Získání počtu záznamů kolekce:

```
Employee employee = (Employee) session.load(Employee.class, new Long(1));  
  
long phonesCount = ((Long) session.createFilter(employee.getPhones(),  
    "select count(*)").list().get(0));
```

- Získání části záznamů kolekce:

```
List<Phone> phones = session.createFilter(employee.getPhones(), "")  
    .setFirstResult(0).setMaxResults(10).list();
```

Batch Fetching I.

- Pomocí batch fetching může Hibernate načíst několik entit najednou. Jedná se o optimalizaci lazy select fetching strategie. Batch fetching můžete zapnout na:

- Třídě
- Kolekci

- **Batch fetching na třídě:**

```
List<Employee> list = session.createQuery("from Employee").list();  
  
for (Employee employee : list) {  
    if (employee.getAddress() != null) {  
        System.out.println("Employee: " + employee.getName() + ", home city: "  
            + employee.getAddress().getCity());  
    }  
}
```

```
}  
  
Zapnutí batch fetching:  
@BatchSize(size=10)  
@Entity
```


Počet záznamů, které se
přednačtou v jednom SELECTu

↖
Vykoná tolik SELECTů,
kolik je tříd typu Employee
v databázi!

Batch Fetching II.

- **Batch fetching na kolekci:**

```
List<Project> list = session.createQuery("from Project").list();  
  
for (Project project : list) {  
    System.out.println("Project: " + project.getName());  
    List<Employee> employees = project.getEmployee();  
    for (Employee employee : employees) {  
        System.out.println(employees.getName());  
    }  
}
```



Vykoná pokaždé jeden SELECT!

Zapnutí batch fetching ve třídě Project:

```
@BatchSize(size=10)  
@ManyToMany(mappedBy = "projects")  
private List<Employee> employees;
```

Batch Fetching III.

- Také můžete zapnout batch fetching globálně:

```
<property name="hibernate.default_batch_fetch_size" value="10"/>
```

2nd level cache

- Existuje několik typů 2nd level cache:
 - **ConcurrentHashMap** – pouze pro testování, výchozí 2nd level cache od Hibernate 3.2, pouze v paměti
 - **EHCache** – Do Hibernate 3.2 výchozí 2nd level cache, může být v paměti, na disku, transakční, podporuje cluster
 - **Infinispan** – podporuje cluster, transakční

2nd level cache – quick setup

- Second level cache je cache na úrovni session factory. Cache nemá tušení o změnách, které byly provedeny jinou aplikací.
- Pro zapnutí 2nd level cache je nutné:

- **Hibernate 4:**

persistence.xml:

```
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
```

pom.xml:

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-ehcache</artifactId>  
  <version>4.1.6.Final</version>  
</dependency>
```

Pokud nepoužíváte persistence.xml, pak je nutné nastavit v hibernate.cfg.xml property: `javax.persistence.sharedCache.mode`

Entitám, které mají být součástí cache je nutné přidat:

```
@Cacheable  
@Entity  
public class Employee {  
  // ...  
}
```

- **Hibernate 3:** <http://ehcache.org/documentation/user-guide/hibernate>

Shared Cache Mode

- Standardně nejsou entity součástí 2nd level cache. Pro nastavení které entity budou její součástí je nutné nastavit shared cache mode, který má následující hodnoty:
 - **ENABLE_SELECTIVE:** výchozí, doporučené nastavení – entity nejsou součástí cache, pokud nejsou explicitně označené jako `@Cacheable`.
 - **DISABLE_SELECTIVE:** entity jsou cachované pokud nejsou označené `@Cacheable(false)`
 - **ALL:** všechny entity jsou cachované
 - **NONE:** žádné entity nejsou cachované (pro kompletní vypnutí 2nd level cache).

Cache strategie

- Je možné blíže upravit strategii přístupu k entitě uvnitř 2nd level cache pomocí anotace `@Cache(usage=CacheConcurrencyStrategy.TYP_STRATEGIE)`.
- Tuto anotaci je také možné nastavit na vazbě uvnitř entity.
- Jednotlivé typy strategií:
 - NONE
 - READ_ONLY
 - NONSTRICT_READ_WRITE
 - READ_WRITE
 - TRANSACTIONAL

Správa cache

- Pro odebrání objektu z 1st level cache je nutné zavolat na objektu typu `Session`:
 - `evict(object)`: Odebere object z 1st level cache.
 - `clear()`: Odebere všechny objekty z 1st level cache.
- Pro odebrání objektu z 2nd level cache je nutné zavolat výše uvedené metody na objektu typu `SessionFactory`.
- `Session` má navíc metodu `contains(object)`, která vrátí `true`, pokud je objekt součástí 1st level cache.

SessionFactory vs. Session

- SessionFactory je expensive-to-create, threadsafe objekt, který je určen k tomu, aby byl sdílen všemi vlákny aplikace. Vytváří se jenom jednou, obvykle při startu aplikace (při integraci se Springem jako Spring bean).
- Session je inexpensive, non-threadsafe objekt, který by měl být použit pro jeden request, konverzaci nebo jednu unit of work.
- Hibernate automaticky vypíná auto-commit mód, všechny změny stavu databáze musí běžet v transakci.
- Je zbytečné otevírat a zavírat session pro každý jednoduchý dotaz nad databází. Sekvence dotazů do databáze se sdružuje do Unit of Work.

Unit of Work

- Unit of Work je série operací, které chceme vyvolat na databázi společně. Prakticky se jedná o logickou transakci, která může vyvolat několik databázových transakcí.
- Nejčastěji se při implementaci Unit of Work setkáte s návrhovým vzorem **session-per-request**. V tomto návrhovém vzoru se pošle request klienta na server, kde běží Hibernate. Otevře se session a veškeré databázové operace se vykonají v této Unit of Work. Na konci, když je připraven response pro klienta, se provede flush a session se uzavře.
- Řada business procesů ale vyžaduje celou sérii interakcí s uživatelem, ve kterých uživatel přistupuje k databázi (například u vyplnění formuláře na více stránek – klasický wizard). Z hlediska uživatele se tato Unit of Work nazývá **long-running conversation**.

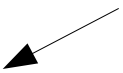
Optimistic concurrency control I.

- Existuje několik způsobů jak implementovat long-running conversation. Nejpoužívanější je **Optimistic concurrency control**.
- Všechny způsoby používají **verzování** v databázi. K entitě je nutné přidat atribut:

@Version

`private long version;`

Většinou je typu long, ale může být:
short, int, long, Timestamp, Calendar



- Hibernate automaticky incrementuje hodnotu tohoto atributu při flush operaci, přičemž když detekuje změnu provedenou jinou session, pak vyhodí výjimku. Je na vývojáři, aby tuto výjimku odchytil a ošetřil (buď nabídne uživateli merge dat, nebo restart business konverzace s aktuálními daty).
- Abyste vynutili kontrolu verze na datech která neaktualizujete, můžete zavolat: `session.lock(object, LockMode.READ);`

Optimistic concurrency control II.

- Hibernate nabízí tři možnosti implementace:
 - **Application version checking**
 - Je možné hlídat verzi ručně, ale je to moc pracné.
 - **Extended session and automatic versioning**
 - Ukončením transakce se session odpojí od JDBC connection. Je možné session někam uložit a později ji otevřením transakce opět připojit k JDBC connection.
 - Problematické, když je session moc velká (session také plní funkci 1st level cache a obsahuje řadu objektů).
 - Známé také jako session-per-conversation
 - **Detached objects and automatic versioning**
 - Každá interakce s persistentním úložištěm probíhá v nové Session, ale používají se stejné persistentní instance při jednotlivých interakcích s databází.
 - Nejpoužívanější strategie.

Optimistic concurrency control III.

```
// vytvorim noveho zakaznika a ulozim ho do databaze
Employee emp = new Employee();
emp.setName("Jirka");
queryService.addEmployee(emp);
// emp.version = 0

// jiny uzivatel si ho mezitim nacetl a zmenil jeho stav
Employee emp2 = queryService2.getEmployee(emp.getEmployeeId());
emp2.setName("Jirka 2");
queryService2.updateEmployee(emp2);
// emp2.version = 1, v databazi se nastavila verze = 1

// kdyz se pokusim provest zmenu do databaze, tak se vyhodi chyba typu
// StaleObjectStateException
emp.setName("Jirka 3");
// emp.version = 0, ale v databazi je verze = 1
queryService.updateEmployee(emp);
```

Logování I.

- Při ladění práce s databází je vhodné nastavit logování. Je to možné provést několika způsoby.
- Nastavení následující property v `persistence.xml` u JPA implementace Hibernate zobrazí všechny SQL dotazy a příkazy, které se posílají do databáze:

```
<property name="hibernate.show_sql" value="true" />
```

- Pro zformátování kódu použijte tuto property:

```
<property name="hibernate.format_sql" value="true" />
```

- Tato property Vám řekne jak se Hibernate k dotazu dopracoval:

```
<property name="hibernate.use_sql_comments" value="true" />
```

- Další konfigurace logování se provádí pomocí logovacího frameworku, viz. další stránka.

Logování II.

- K logování se obvykle používá knihovna Log4j. Pro její zapnutí je nutné přidat do pom.xml následující dependency:

```
<dependency>  
  <groupId>org.slf4j</groupId>  
  <artifactId>slf4j-log4j12</artifactId>  
  <version>1.6.6</version>  
</dependency>
```

Logování III.

- Následně přidejte do classpath soubor log4j.properties s následujícím obsahem:

```
log4j.appender.file=org.apache.log4j.RollingFileAppender
```

```
log4j.appender.file.File=hibernate.log
```


```
log4j.appender.file.MaxFileSize=1MB
```

```
log4j.appender.file.MaxBackupIndex=1
```

```
log4j.appender.file.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

Bude vypisovat logovací hlášky
do souboru hibernate.log




```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.stdout.Target=System.out
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

Bude vypisovat logovací
hlášky na konzoli





```
log4j.rootLogger=INFO, file, stdout
```

Zapojení výše uvedených konfigurací



```
log4j.logger.org.hibernate=INFO
```

Úrovně závažnosti chyb,
které se budou logovat



```
log4j.logger.org.hibernate.type=ALL
```

Logování IV.

- Všechny logovací levely:
 - http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html_single/#configuration-logging

JPA Callbacks

- Je možné definovat metody s jednou z anotací:
 - `@PrePersist` (před operací `persist()`), `@PostPersist` (po operaci `persist()`)
 - `@PreRemove`, `@PostRemove`
 - `@PreUpdate`
 - `@PostLoad`
- Tyto metody je možné definovat buď v entitě, nebo v jiné třídě, která se přes anotaci `@EntityListeners` propojí s vybranou entitou.
 - http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#events-jpa-callbacks

Interceptors I.

- Interceptory umožňují aplikaci manipulovat atributy persistentního objektu před jeho uložením, update, smazáním nebo načtením z databáze.
- Vytvořte třídu Auditable a přidejte k definici třídy Employee extends Auditable:

```
@MappedSuperclass
public class Auditable {

    @Column(name="CREATE_DATE")
    private Date createDate;

    public Date getCreateDate() {
        return createDate;
    }

    public void setCreateDate(Date createDate) {
        this.createDate = createDate;
    }
}
```

```
@Entity
public class Employee extends Auditable {
    // ...
}
```

Interceptors II.

- Vytvořte interceptor:

```
public class AuditableInterceptor extends EmptyInterceptor {  
    @Override  
    public boolean onSave(Object entity, Serializable id,  
                          Object[] state, String[] propertyNames, Type[] types) {  
        if (entity instanceof Auditable) {  
            for (int i = 0; i < propertyNames.length; i++) {  
                String propertyName = propertyNames[i];  
                if ("createDate".equals(propertyName)) {  
                    state[i] = new Date();  
                    return true;  
                }  
            }  
        }  
        return false;  
    }  
}
```


Interceptors III.

- Interceptor je možné zapojit:

- **Na úrovni session:**

- Session session = sessionFactory.openSession(new AuditInterceptor());

- **Na úrovni session factory** – existuje několik možností zapojení:

- Zapojení v persistence.xml:

```
<property name="hibernate.ejb.interceptor"
          value="com.test.entity.interceptor.AuditableInterceptor"/>
```

- Zapojení ve Spring konfiguraci (pouze pro Hibernate 3):

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="entityInterceptor">
        <bean class="com.test.entity.interceptor.AuditableInterceptor"/>
    </property>
</bean>
```

Events

- V Hibernate existuje systém událostí, který je možné použít současně s interceptory, nebo jako jejich náhradu.
- Události mění defaultní chování metod.

SchemaExport I.

- Pokud používáte Spring a Hibernate (pomocí JPA), pak můžete vygenerovat databázové schéma pomocí třídy SchemaExport následovně:

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@ContextConfiguration("classpath:db-dev.xml")
```

```
public class SchemaExportTest {
```

```
@Autowired
```

```
private LocalContainerEntityManagerFactoryBean entityManagerFactory;
```

```
@SuppressWarnings("deprecation") @Test
```

```
public void exportDatabaseSchema() {
```

```
    PersistenceUnitInfo persistenceUnitInfo = entityManagerFactory.getPersistenceUnitInfo();
```

```
    Map<String, Object> jpaPropertyMap = entityManagerFactory.getJpaPropertyMap();
```

```
    Configuration configuration
```

```
        = new Ejb3Configuration().configure(persistenceUnitInfo, jpaPropertyMap).getHibernateConfiguration();
```

```
    SchemaExport schema = new SchemaExport(configuration);
```

```
    schema.setOutputFile("gen-schema.sql");
```

```
    schema.create(false, false);
```

```
}
```

```
}
```

↖ Připojení do databáze pomocí
LocalContainerEntityManagerFactoryBean

SchemaExport II.

- Několik poznámek k předcházejícímu kódu:
 - Konfigurace připojení do databáze je v souboru db-dev.xml, který je v classpath a používá se v něm třída `LocalContainerEntityManagerFactoryBean`
 - Výsledný skript se uloží do domovského adresáře projektu do souboru `gen-schema.sql`
 - Tento kód funguje pro Hibernate do verze 5 (která doposud nevyšla. V této verzi Hibernate už nebude třída `Ejb3Configuration`, která se v tomto příkladě používá)
 - Metoda `create` má dva parametry typu `boolean`:
 - Jestli bude skript vypisovat DDL do konzole
 - Jestli se bude DDL skript exportovat do databáze